Advances in Artificial Intelligence and Machine Learning; Research 3 (4) 1526-1556 Received 23-05-2023 ; Accepted 15-10-2023 ; Published 22-10-2023

A Journey Towards the Most Efficient State Database For Hyperledger Fabric

Ivan Laishevskiy ivan.laishevskii@scientificideas.org
Idea Blockchain Research Lab
Moscow, Russia

Artem Barger bartem@scientificideas.org
Idea Blockchain Research Lab
Haifa, Israel

Vladimir Gorgadze gorgadze.vv@mipt.ru
Moscow Institute of Physics and Technology
Moscow, Russia

Corresponding Author: Vladimir Gorgadze

Abstract

Hyperledger Fabric is a leading permissioned blockchain platform known for its flexibility and customization. A crucial yet often overlooked component is its state database, which records the current state of blockchain applications. While the platform currently supports LevelDB and CouchDB, this study argues that there is an unmet need for exploring alternative databases to enhance performance and scalability. We evaluate RocksDB, Boltdb, and BadgerDB under various workloads, focusing on memory and CPU utilization. Our findings reveal that each alternative outperforms the existing options: RocksDB excels in throughput and latency, Boltdb minimizes CPU usage, and BadgerDB is most memory-efficient. This research not only provides a roadmap for integrating new state databases into Hyperledger Fabric but also offers critical insights for those aiming to optimize enterprise blockchain systems. The study underscores the significant gains in scalability and performance that can be achieved by reconsidering the choice of state database.

Keywords: Blockchain, Key value store, Database, LevelDB, RocksDB, BadgerDB, BoltDB, Hyperledger fabric, Hyperledger caliper, State database.

1. INTRODUCTION

In recent years, the popularity of blockchains as a means of payment and asset transfer has increased significantly. The public perception of blockchain technology has evolved from initial unawareness and suspicion to current optimism about the potential for decentralized asset management [1]. The technology is also expanding on its use cases, facilitating intermediation among mutually untrusted

Citation: Ivan Laishevskiy, et al. A journey towards the most efficient state database for Hyperledger Fabric Advances in Artificial Intelligence and Machine Learning. 2023;3(4):88.

entities. A blockchain is a shared and distributed ledger of transactions that all network participants maintain multiple copies of, providing secure and transparent record-keeping. Transactions are grouped into hash-chained blocks, with each block using a cryptographic hash function pointing to its immediate predecessor, ensuring the immutability of records.

Blockchain technology has been gaining momentum in recent years, with its potential to disrupt various industries and transform the way businesses operate. As such, there has been a surge in the adoption of blockchain-based solutions in various sectors, including finance, healthcare, and supply chain management, among others. This trend is supported by the increasing number of studies and research works that explore the potential of blockchain technology, as cited by [2–4]. Among the many blockchain frameworks available today, Hyperledger Fabric has emerged as a popular choice for businesses looking to implement blockchain-based solutions. This is due to its flexible architecture, which allows for the creation of customized solutions tailored to meet specific business needs, as well as its robust security features and scalability. Additionally, Hyperledger Fabric has a vibrant community of developers, making it easy for businesses to find skilled professionals to build and deploy blockchain solutions. This is supported by several research works, as cited by [5–7].

Hyperledger Fabric is an open-source initiative designed to meet the specific needs of businesses and plays a significant role in the development of enterprise-grade solutions [8]. Fabric offers an enterprise-grade, permissioned blockchain platform and leverages the execute-order-validate paradigm for smart contract execution, ensuring that transactions are executed in a deterministic order and verified by the network [9, 10]. Its modular architecture and customizable components make it a reliable and cost-effective choice for businesses developing blockchain solutions [5].

Businesses can develop tailored solutions that meet their specific needs by leveraging Fabric's components, enabling integration with existing systems and applications, reducing development costs and time to market [5]. Furthermore, the open-source nature of Hyperledger Fabric has attracted a vibrant community of developers, researchers, and industry leaders, who work together to enhance and expand the platform's capabilities. This collaborative effort has resulted in a reliable and future-proof platform that meets the needs of businesses across various industries.

The open-source nature of Hyperledger Fabric has contributed significantly to its popularity and trustworthiness among businesses. Fabric's source code is available for review, enabling enterprises to contribute to its development and customize it to meet their specific needs. This feature makes it a cost-effective and flexible choice for businesses, and its pluggable architecture ensures that it can be integrated with various systems and applications. Additionally, the vibrant community of developers and contributors has worked together to enhance and expand the platform's capabilities, resulting in a platform that meets the needs of businesses across various industries and has significant potential to drive innovation and transformation in the enterprise sector.

The state database, or StateDB, is an essential component of the Hyperledger Fabric blockchain platform that plays a crucial role in maintaining the integrity of the ledger. It captures the most up-to-date snapshot of the world state and accumulates blockchain transaction updates into a persisting key-value database [9].

During the execution of transactions, peers rely on StateDB to read the current values with their version to form the Read-Set for transactions. Then, during the commit and validation phase, while executing the multi-value concurrency control check, peers read from StateDB to compare Read-

Set versions [11]. Once the validation is complete, the peer commits validated transactions into StateDB to reflect the recent changes, ensuring the integrity of the ledger.

The StateDB plays a crucial role in maintaining the integrity of the Hyperledger Fabric blockchain. Peers rely on the StateDB to read the current values, and it accumulates blockchain transaction updates into a persisting key-value database. The database is used during the commit and validation phase, where peers read from StateDB to compare Read-Set versions, commit validated transactions into StateDB to reflect recent changes, and ensure the integrity of the ledger. The use of StateDB in Hyperledger Fabric provides a reliable and secure platform for businesses developing blockchain solutions.

According to the [12], the commit phase in HLF dominates the transaction processing time and, as a result, constitutes a bottleneck in terms of performance. In addition, the study demonstrates that the read operation plays an important role when peers replicate transactions. Clearly, the interaction with the StateDB directly affects the overall performance of the HLF. Currently, HLF presents two potential StateDB implementations, one based on GoLevelDB and the other on CouchDB. Therefore, the Fabric community acknowledged the necessity to develop a superior alternative to StateDB [13]. In this paper, we will analyze various options for StateDB and examine the various implementations of the embedded databases based on LSM-trees or B+ trees [14, 15].

This paper is structured in the following. The background section, which delves into the key components and concepts of Hyperledger Fabric, including peers, the ordering service, and StateDB. The Related Work section reviews existing literature and studies that have explored the performance aspects of Hyperledger Fabric and its state database. In the Methodology section, we detail our experimental setup and the benchmarks employed to assess performance. The Results and Analysis section presents our findings, offering a nuanced view of how different databases impact the performance metrics of Hyperledger Fabric. Subsequently, the paper identifies Opportunities for Low-Effort, High-Impact Improvements, pinpointing specific areas where resource utilization and throughput can be optimized. The Discussion section elaborates on the implications of our findings, suggesting potential avenues for further research and optimization. Finally, the Conclusion summarizes the key insights gained from this study, offering a cohesive wrap-up of our contributions to the field of enterprise blockchain technology.

2. BACKGROUND

2.1 Hyperledger Fabric

The Hypereldger Fabric (HLF) blockchain platform is a permissioned system with a flexible and modular design. It ensures a high anonymity level while still being stable and scalable. The permissioned aspect of the platform is achieved using an abstraction known as a Membership Service Provider. This abstraction encapsulates the identity management and credentials of businesses engaging with the platform. The Membership Service Provider ensures that all entities engaged in the transaction coordination formation have their authenticity and authorization verified.

The concept of channels is brought into play by Fabric architecture to facilitate improved data segregation, which protects users' privacy and enables sharding, alleviating scalability issues. In addition,

the information may be safely traded between consortium members through the channel formed by the parties who have organized themselves into a consortium. Channels are a straightforward abstraction of the private blockchain instance's ledger management, smart contract administration, and other features. The nodes that participate in the Hyperledger Fabric blockchain network typically fulfill one of the following three roles:

- The business application code is tasked with coordinating the flow of transaction processing, and it typically uses the Fabric SDK to interact with the Hyperledger Fabric.
- The endorsing and committing peers are responsible for managing the state, validating transactions, and providing attestation of agreement of transaction execution results.
- Ordering service instituting atomic broadcast abstraction to carry out consensus protocol, which is used to group client transactions into blocks and guarantee total order between all transactions.

The following is an example of how the transaction processing flow of Hyperledger Fabric may be described:

1. The client is responsible for putting together a signed transaction and submitting the proposal to the group of peers supporting it per the endorsement policy.
2. Each endorsing peer calls chaincode to simulate transactions, recording state updates as read-writes to capture all state-related operations. This is done by recording read-writes and state updates. After that, sending endorsed peer-computed hashes of the read-write set together with the signed hashed result will result in the client receiving both the read-write sets and the signed hash.
3. The client is responsible for organizing the answers from all the endorsing peers and validating that all the endorsing peers have signed the same payload. After that, it constructs a transaction by fusing all of the signatures of the peers who endorsed it together with the read-write sets.
4. The client invokes an "atomic broadcast" application programming interface (API) to transmit the transaction to the ordering service. For instance, the client will only place one order while utilizing the Raft-based ordering service.
5. The ordering service is responsible for gathering all incoming transactions, grouping them into blocks, and obtaining approval on the blocks to ensure the transactions are in the correct order.
6. Peers communicate with one another to distribute blocks using a process based on gossip. This allows certain peers to obtain blocks directly from the ordering service while others share blocks.
7. When a peer receives a new block, they loop over their transactions and validate
   (a) the endorsement policy and
   (b) execute multi-version concurrency control checks against the state. This happens after they have received the new block.

8. Once peers have validated the transactions, the block will be attached to the distributed ledger, and the state will be updated to reflect any recent changes.

When the block is finally committed, the peer sends out an event to alert any interested clients.

2.2 State Database

In Hyperledger Fabric, the state database is a key-value store that is used to track the current state of the ledger. It stores the current values of all assets on the ledger, as well as the current state of any smart contracts that have been deployed on the platform. The state database is updated whenever a transaction is committed to the ledger, and it is used to validate the state of the ledger before a transaction is committed. The state database in Hyperledger Fabric is a key-value store used to track the current state of the ledger. The state database is designed to be scalable, reliable, and fast, supporting the high transaction volumes and performance requirements of enterprise-grade blockchain applications.

The state database is implemented using GoLevelDB, which is an open-source, on-disk key-value store that is optimized for fast, read-heavy workloads. GoLevelDB is known for its simplicity, reliability, and performance, making it a good choice for the state database in Hyperledger Fabric. Hyperledger Fabric also provides an option to use Apache CouchDB as the state database. CouchDB is an open-source, NoSQL database that uses a document-oriented data model and is known for its ease of use and scalability. One key advantage of using CouchDB as the state database in Hyperledger Fabric is its ability to store data in a JSON format, which makes it easy to work with and integrate with other systems. CouchDB also provides a rich set of indexing and querying capabilities, allowing developers to easily retrieve and manipulate data stored in the state database.

A global state database takes a snapshot of the state based on the most recent set of valid transactions, and the ledger structure of Hyperledger Fabric is similar to that of other blockchain platforms in that it is an append-only series of hash-chained blocks. To facilitate execution and check the read-write-set against the real state value, the peers committing transactions and approving transactions use the state database to read keys and supply values into chaincode.

Let's examine the impact of StateDB on HLF performance. For this, a number of performance benchmarks will be run, and in some cases, the HLF profiles will be examined. In this study we would focus only on analysis of GoLevelDB implementation of the state database, as it was shown that it outperforms CouchDB [16]

2.3 GoLevelDB

GoLevelDB is an implementation of LevelDB in the Go programming language. LevelDB itself is a fast key-value storage library that provides an ordered mapping from string keys to string values. This section aims to delve into the actual details of LevelDB, its design architecture involving Log-Structured Merge-trees (LSM trees), and how it differentiates from other databases. We will also discuss the types of applications for which GoLevelDB is most suitable.

LevelDB utilizes a Log-Structured Merge-tree (LSM tree) to achieve high write throughput and efficient range queries. Unlike traditional B-trees used in relational databases, LSM trees are optimized for write operations. The LSM tree consists of a memory component, known as the MemTable, and multiple levels of SSTables (Sorted String Tables) on disk.

The LSM tree architecture offers several advantages:

1. **High Write Throughput:** The LSM tree is optimized for write operations, which means that it can handle a large number of writes per second. This makes it ideal for applications that require high write throughput, such as logging and analytics.
2. **Efficient Disk Reads** The LSM tree is optimized for range queries, which means that it can handle a large number of reads per second. This makes it ideal for applications that require high read throughput, such as search engines and social networks.
3. **Write Amplification Minimization** The LSM tree minimizes write amplification by using a log-structured merge-tree (LSM tree) to store data on disk. This means that it can handle a large number of writes per second without incurring significant overhead.

GoLevelDB leverages the LSM tree architecture to offer high write throughput and efficient disk reads, making it a suitable choice for specific types of applications that do not require the complexities of a full-fledged relational database. Its design makes it distinct and optimized for write-heavy workloads, although it comes with the trade-off of limited query capabilities.

2.4 CouchDB

CouchDB is a NoSQL database that uses a schema-free JSON document format to store data. Unlike GoLevelDB, CouchDB is designed with a focus on ease of use and being "a database that completely embraces the web." This section will explore the architecture of CouchDB, its design principles, and how it differs from databases like GoLevelDB. We'll also discuss the types of applications for which CouchDB is most suitable.

CouchDB uses a Multi-Version Concurrency Control (MVCC) system to manage concurrent reads and writes. It also employs a B-tree indexing model, which is different from the LSM trees used in LevelDB.

1. **Multi-Version Concurrency Control (MVCC)** CouchDB uses a Multi-Version Concurrency Control (MVCC) system to manage concurrent reads and writes. This means that it can handle a large number of reads and writes per second without incurring significant overhead.
2. **B-Tree Indexing Model** CouchDB uses a B-tree indexing model, which is different from the LSM trees used in LevelDB. This means that it can handle a large number of reads and writes per second without incurring significant overhead.
3. **Schema-Free JSON Document Format** CouchDB uses a schema-free JSON document format to store data. This means that it can handle a large number of reads and writes per second without incurring significant overhead.

CouchDB offers a different set of features and optimizations compared to GoLevelDB. Its architecture is designed to be web-friendly and to offer strong consistency and availability guarantees. While it may not be as write-optimized as databases that use LSM trees, it offers a rich set of features that make it suitable for a wide range of applications, particularly those that are read-heavy or require complex queries.

2.5 State Database Configuration

GoLevelDB comes pre-packaged with Hyperledger Fabric, so no additional installation steps are required. To initialize a network using GoLevelDB, you can use the standard network initialization scripts provided by Hyperledger Fabric. No changes are needed in the configuration files when using GoLevelDB, as it is the default state database. However, if you want to use CouchDB instead of GoLevelDB, you will need to make some changes in the configuration files. The following steps will show you how to configure CouchDB as the state database in Hyperledger Fabric.

1. Install CouchDB on your machine. You can download CouchDB from the official website of Apache CouchDB.
2. Open the core.yaml file and make the following changes:
   (a) Set the port number to 5984.
   (b) Set the bind address to address of the couchdb container.
   (c) Set the admin username and password.

Here is the example of the configuration file:

```
ledger:
state: CouchDB
couchDBConfig:
couchDBAddress: <CouchDB_Container:Port>
username: <Admin_Username>
password: <Admin_Password>
```

The key difference in two configurations is that CouchDB is a document-oriented database, while GoLevelDB is a key-value store. This means that CouchDB stores data in the form of documents, while GoLevelDB stores data in the form of key-value pairs. This difference in data storage format has a significant impact on the performance of the state database.

2.6 Performance Baseline

There is a need to establish a starting point for comparison and therefore we implemented a benchmark to identify the maximum potential performance improvement that could be achieved by replacing the current implementations with more performant alternatives. In addition, it serves as a

baseline comparison to detect the potential performance gains that may be possible. The benchmark implements chaincode FixedAsset with two methods, one which results in interaction between the peer and the state database and the other which does not.

In this paper, similarly [17], the term ”transaction” is used in a general sense and refers to any interaction with a smart contract, regardless of the complexity of the subsequent interaction(s) with the HLF platform. In this article, we will give an extra description and route map for each type of transaction that is employed (e.g. as in FIGURE 1).

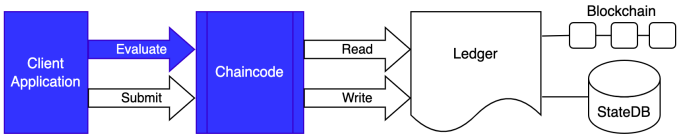

Figure 1: Diagram of HLF components interaction when running the Empty-Contract-Evaluate benchmark. Active components are highlighted in blue.

The chaincode benchmark that does not interact with StateDB consists of loads of Evaluate and Submit calls to the EmptyContract function of the FixedAsset smart contract written in Golang within the Hyperledger Fabric network, where goleveldb is used as StateDB.

EmptyContract Evaluate transactions will being run on a single Hyperledger Fabric peer and will not result in any interaction with the Orderer, resulting in the transaction pathway depicted in FIGURE 1. Thereby, there is no interaction with StateDB at this stage. Compared to this, we can understand how much longer the work of a smart contract takes if something needs to be read from StateDB.

EmptyContract Submit transactions will being run on a single Hyperledger Fabric peer and then, after approval by the Orderer, are written to the blockchain. It will result in the transaction pathway depicted in FIGURE 2.

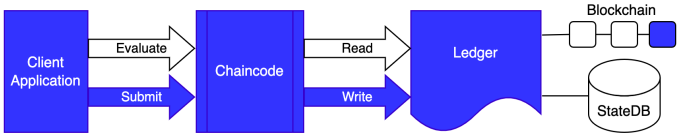

Figure 2: Diagram of HLF components interaction during the Empty-Contract-Submit benchmark. Active components are highlighted in blue.

Thereby, there is no interaction with StateDB at this stage. Compared to this, we can understand how much longer the work of a smart contract takes if something needs to be written to StateDB.

Achievable throughput is investigated through maintaining a constant transaction backlog of 100 transactions for each of the 10 test clients.

To compare Evaluate of the EmptyContract function of the FixedAsset chaincode, Evaluate of the GetAsset function of the same smart contract is called. The key value length is 100 bytes (detailed

description is given in the section 6.1). To compare Submit of the EmptyContract function, Submit of the CreateAsset function of the same smart contract is called (detailed description is given in the section 6.2).

We can conclude that reading small values in transactions from StateDB has essentially no impact on HLF performance compared to writing the same values to StateDB based on the findings of comparing the average TPS in FIGURE 3.

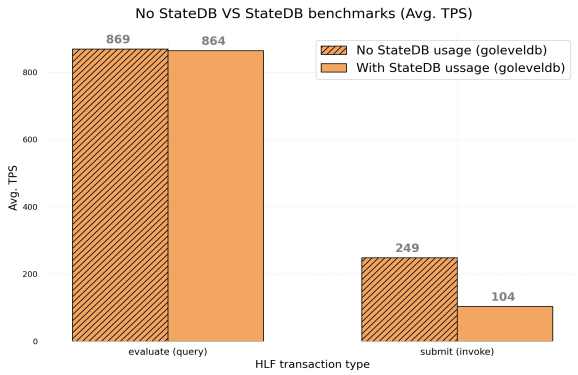

Figure 3: NoStateDB-VS-StateDB performance benchmark results.

As a result, in this study, we gave top importance to enhancing StateDB's write performance.

2.7 Profiler Analysis

We utilized a profiling tool [18] to obtain the execution footprint of the Hyperledger Fabric peer node to detect potential bottlenecks for improvements and optimizations. The profiling data collection performed on transactions updating or adding a piece of new information into state database because it was indicated earlier 2.6 write operation has more impact on overall HLF performance compared to read operations.

While conducting profiling on HLF configured with goleveldb-based implementation of the state database, we noted that while peer nodes were performing an update of transactions applying them into the state, the peer spent only roughly 0.32% of CPU time. However, continuing profiling [19], we discovered that 6% of the overall profile sampling time was spent off-CPU, waiting for additional IO and syscalls. Furthermore, further tracing [20] has shown that time was spent during syscall.Fsync, meaning transferring data from RAM into actually the disk.

In addition, it was discovered that the HLF peer uses 9 additional distinct goleveldb entities in addition to StateDB. Every one of which starts 5 auxiliary goroutines that are waiting for compression, looking for compression errors, etc. On the basis of this, it can be concluded that the routines for waiting for goleveldb events are redundant in the HLF peer.

As a result, StateDB (and goleveldb behind the scenes) are crucial to the functionality of Hyperledger Fabric and have a big impact on write operation performance.

3. OPPORTUNITIES FOR LOW-EFFORT, HIGH-IMPACT IMPROVEMENTS

We analyzed resources to use during the loading of peers and identified a few areas for improvement. Inspecting the findings depicted at2.7, we saw that even though disk IO can handle 30M/s, we only achieved 14M/s with 45 tps and just 27% CPU use. With an increase in load to 100 tps, we only utilized 10% of our CPU and obtained 6M/s. These observations suggest that resources should be used to their maximum potential; consequently, identifying the bottlenecks might result in significant changes that increase throughput.

The trace analysis showed an excessive wait in the blocked mutex after each block delivery and synchronization in the function responsible for delivering blocks. We also found evidence in the code [21], explaining that while committing a block, the client can get a message confirming the new block's successfully committed while also receiving the old state. After carefully analyzing the code, we found no risk to eliminating the lock [22]. We ran all integration and unit tests supplied by Fabric to ensure no regression was made and results revealed no errors, while TPS, CPU utilization, and disk write speed all showed improvements. In FIGURE 4, the outcome of modifications in the utilization of CPU and TPS resources for various types of transactions is depicted. FIGURE 5 shows a difference in disk write speed and TPS.

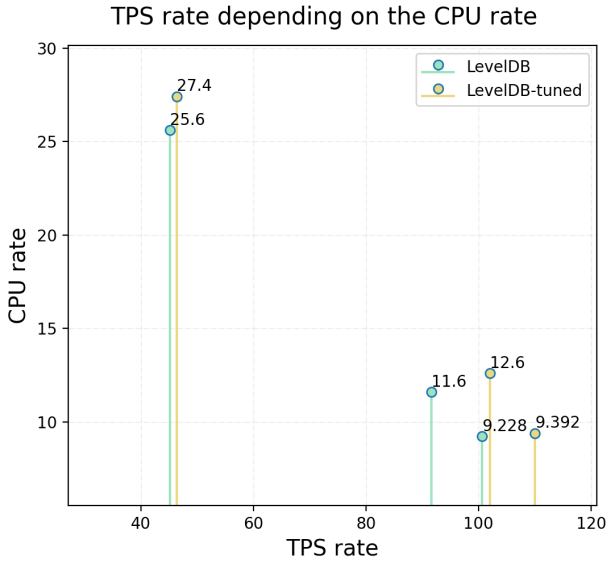

Figure 4: Comparison of TPS versus CPU usage before and after eliminating excessive mutex lock waits. For three types of transactions, 64KB, 4KB and 100B in size (left to right in the graph), improvements were found both on the OX axis and on the OY axis. "LevelDB" means HLF with StateDB goleveldb before code changes, and "LevelDB-tuned" means HLF with StateDB goleveldb after code changes.

Our implementation resulted in more efficient resource utilization and improved performance of HLF. We also conclude that there is potential for further identification and optimization of bottlenecks in the existing HLF code.

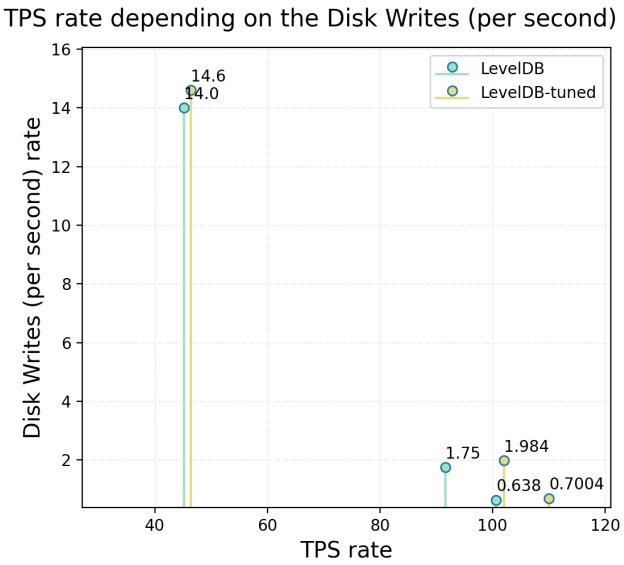

Figure 5: Comparison of TPS versus disk write rate before and after eliminating excessive mutex lock waits. For three types of transactions, 64KB, 4KB and 100B in size (left to right in the graph), improvements were found both on the OX axis and on the OY axis. ”LevelDB” means HLF with StateDB goleveldb before code changes, and ”LevelDB-tuned” means HLF with StateDB goleveldb after code changes.

4. Alternative Statedbs Selection

4.1 Criteria for Choosing an Alternative StateDB

It turns out that it makes sense to consider other databases for StateDB HLF. The following main criteria were used to choose the databases:

- Support for key-value format is needed to maintain compatibility across all HLF components and cut DB embedding time;
- The availability of an API in Golang or the implementation of storage in Golang;
- There is documentation available;
- The accessibility of open access performance benchmark results that set this key-value store apart from others;
- Desirably, the storage should provide a lock to make changes when the database is opened by another process. As a result, handling process blocking in HLF code won’t be necessary in the future. Today this is artificially enforced in HLF through the `FileLock` method described in 5.1.3.
- Additionally, the storage should provide thread safety. Because today in HLF this is guaranteed by locking the mutex on every database operation.

- Of course, storage with ACID offers the highest level of assurance for atomic change, data consistency, transaction isolation, and durability;
- The store must include iteration capability since HLF employs iterators;
- The batch (a set of keys and values) writes mechanism must also be supported in an alternative StateDB due to the fact that HLF primarily uses batch writes;
- To support the functionality of client queries in HLF, today it is necessary to connect an additional StateDB to the peer - CouchDB [23]. In this context, it would be advantageous if the goleveldb substitute allowed for database queries.
- Built-in logical storage partitioning mechanisms will eliminate the need for logical database partitioning in the HLF code. This will make the project code simpler and, in some situations, let you select the performance-optimal technique. Some databases offer various logical partition types to this goal. Today, in HLF, logical storage separation is supported by the `DBHandle` structure described in 5.1.2.
- A rough estimate of connection complexity;
- The projected difficulty of sustaining database functioning, accounting for future HLF upgrades.
- The kind of tree the store uses.

The Table 1 presents distinctive comparison properties of the chosen DBs. A more detailed description of each database is provided below.

4.2 Selecting RocksDB

- Language and brief background. According to [24], RocksDB [25] is the most used database in blockchain. RocksDB is a fork of an earlier version of LevelDB. It is also based on the LSM-tree and is written in C++, like the original LevelDB.

  At the time of writing this article there are several APIs in Golang which allow to work with C++ implementation of RocksDB. Finally the API in the grocksdb [26] repository was chosen as the most popular and active at that time.
- Performance. When writing batches in 2020 in [27] RocksDB demonstrated throughput that was three times greater than goleveldb, but it lost severely in read operations (for an SSD with fsync enabled). Among the obvious differences between RocksDB and LevelDB, RocksDB developers mention several performance optimizations and additional options for interacting with the DB [28].
- Working with multiple processes and threads, ACID. When multiple processes access the RocksDB database, only one of them will be able to write. The DB must be used in read-only mode by the remaining processes. Basic database operations: Put(), Write(), Get(), NewIterator(), as well as the use of constant ReadOptions and WriteOptions are thread-safe. Some of the ACID properties are warranted by the user. For example, such as atomicity in the case of writing batches to several DBs or transaction isolation in the case of using the TransactionDB entity instead of OptimisticTransactionDB.

Table 1: Selected characteristics of key value storages

№	DB	Implementation Language	Documentation availability	Advantage over other DBs in reads	Advantage over other DBs in writes	Reading from multiple processes	Thread safety	ACID	Iterator support	Batch writes support	Queries support	Logical DB partition	Estimated level of embedding difficulty	Estimated level of further DB support difficulty	Tree type
1	goleveldb	Go	pkg.go.dev only	RocksDB	No data	Possible even with one running process in read-write mode	DB object is thread-safe	–	+	+	–	–	–	–	LSM
2	RocksDB	C++	detailed github wiki	No data	goleveldb	Possible even with one running process in read-write mode	Basic operations: Put(), Write(), Get(), NewIterator() are thread-safe. ReadOptions and WriteOptions can be used from different threads	User controlled	+	+	–	Separated DB files, ColumnFamilies	High	High	LSM
3	bbolt	Go	pkg.go.dev only	RocksDB BadgerDB	goleveldb	Possible, but processes in read-write mode will be blocked	Extracting data and creating objects from a database instance, but not operating on derivated objects (e.g. transactions)	+	There are no iterators, but there are cursors	There are no batches, but there are transactions	–[a]	Buckets	Low	Low	B+
4	BadgerDB	Go	dgraph.io/docs/badger pkg.go.dev discuss.dgraph.io	No data	RocksDB bbolt	Only possible if all processes have opened the database in read-only mode	DB object is thread-safe; Transaction object is not thread-safe	+	+	+	–[b]	–	Low	Low	LSM

[a]Possible with additional tool [29] [b]Possible with additional tool [30]

- Tools provided by the storage. RocksDB, like goleveldb, provides iterators. However, it has a slightly modified logic (setting boundaries, closing the iterator, etc.). Multiple keys can be written simultaneously in batches. There is no way to use queries. It was discovered that RocksDB may be logically separated into different databases and Column Families. Each sort of division has distinct qualities.
- Evaluation of the complexity of embedding and further support. The fact that this DB requires an additional API connection with the C++ library, in contrast to others, means that embedding it will take longer than any other DB. Additionally, RocksDB presents difficulties for further support within HLF. One explanation is that each upgrade to the RocksDB version necessitates an API update, followed by a new API and RocksDB library binding. In addition, even as of the time of authoring the paper, several features of the current database version are not implemented in the API (some options are not available, etc.).

The choice was made to add RocksDB as StateDB in HLF due to the variety of features and competing reported performance figures.

4.3 Selecting bbolt

- Language and brief background. As a database with the mechanism of B-tree, or rather its subspecies B+ tree, sometimes mentioned LMDB [31], but the repository with this DB is read-only [32], and in addition it is implemented in C. The current Golang-APIs, which have not been updated since 2017, have limited interfaces and somewhat low-level package bindings [33]. As a result, it was decided not to take LMDB into consideration. However, an alternative database in the Golang language was found - BoltDB, which is “LMDB-inspired” and is also based on the B+ tree [34]. However, the repository’s data indicates that there haven’t been any modifications since 2018. The developers advise utilizing the bbolt fork [35], which is supported as of this writing, for those who desire updates.
- Performance. According to [27], bbolt outperforms goleveldb in batch write transaction throughput by a factor of 3 and exceeds rocksdb in read operations by more than a factor of 2 (for an SSD with fsync enabled).
- Working with multiple processes and threads, ACID. bbolt can run in read-only mode from multiple processes, but does not support shared database access for processes attempting to open a file in read-write mode. In read-write mode, a process will halt until all others have completed. Although the bbolt database object is thread-safe, objects derived from it (such transactions, buckets, and keys) are not. It is claimed that BoltDB supported all ACID properties. bbolt must therefore support ACID if it is to be considered a ”more featureful version of Bolt.”
- Tools provided by the storage. The cursor serves as the bbolt equivalent of the iterator. Only when a transaction is open can all of the cursor’s components be accessed. There is no way to write a batch containing multiple keys, but you can write a pre-opened transaction to which keys have been added. There is no support for queries, but it is possible with the additional tool [29] (although the last update in the repository at the time of writing is 2020). The bbolt database is logically separated into buckets that can be nested inside of one another.

- Evaluation of the complexity of embedding and further support. Given the equivalents of the components that already exist in bbolt for goleveldb and its Golang implementation, connecting it does not appear to be difficult. It appears straightforward to support bbolt functionality in updating HLF. But if HLF is built with the obsolete Extras API [29], things could get trickier.

As a result, it was chosen to implement bbolt as StateDB into HLF without adding any additional APIs.

4.4 Selecting BadgerDB

- Language. BadgerDB is implemented in Golang. The data storage model is an LSM-tree.
- Performance. On DB performance measures, several published studies have been evaluated. Performance tests for HDD and SSD with and without fsync were conducted in [36]. Since an SSD with fsync enabled is the most common type of HLF production peer, we only consider that scenario.

  The updated results for [36], presented in [27], show that in random read operations, pogreb [37] leads by a good margin, followed by buntdb [38], nutsdb [39] and goleveldb, followed by bolt, bbolt and BadgerDB [40]. For single write operation, RocksDB and pebble [41], as well as BadgerDB have performance advantages over goleveldb, although goleveldb showed better results compared to bolt and bbolt. BadgerDB is the leader with a breakaway for the batch writing operation, followed (in descending order) by buntdb, RocksDB, bbolt, pebble, boltdb.

  In a 2017 paper from the BadgerDB developers, [42] for a performance benchmark of writing keys with corresponding values of 128 and 1024 bytes, BadgerDB showed more than a 10-fold and 14-fold advantage over LMDB and boltdb. And in another article [43] the authors noted lower memory consumption of BadgerDB compared to RocksDB, and BadgerDB came out as the leader in writing keys with sizes of their respective values 1024 and 16384 bytes (more than 4 times and 11 times respectively), but nevertheless in performance benchmarks of random read operations RocksDB looked better by more than 3 times.
- Working with multiple processes and threads, ACID. Reading from the database is feasible when it has been opened in read-only mode by all processes. However, if a read-write process has locked the BadgerDB directory, reading from other processes won't function. The database object itself is thread-safe, but transaction objects are not thread-safe. The DB provides all ACID properties.
- Tools provided by the storage. BadgerDB supports iterators and batches. With additional tools [30] it is possible to use queries. BadgerDB does not support logical DB partitioning.
- Evaluation of the complexity of embedding and further support. Given the aforementioned, embedding and maintaining BadgerDB looks simple. It was decided to postpone the use of the API with queries until the future works.

BadgerDB was chosen to be embedded as StateDB because to its competitive features.

The most crucial characteristics of the selected DBs are contrasted in the Table 1.

4.5 A Few Comments About Other Key-Value Storages

The official pogreb repository [37] presents performance benchmark results showing that this database is several times faster in single random read operations than goleveldb, BadgerDB and bboltdb.

In [44], which describes RoseDB [45], performance benchmarks show almost the same write speed as leveldb, which is 2.5 times faster than BadgerDB. Because RoseDB can store keys and values in RAM (if they are not large), its read operation is faster than both goleveldb and BadgerDB, but for the case of storing only keys in RAM (for large values) RoseDB shows approximately equal results with goleveldb and 1.25 times faster than BadgerDB.

However, RoseDB appears to be insufficiently documented, thus there is a chance that using it in HLF will be challenging. Future work may focus on pebble because of its strong single write performance and pogreb because of its highly intriguing read benchmarks, but its repository is not particularly active at the time of paper writing.

5. ALTERNATIVE STATEDBS INTEGRATION

5.1 Components Requiring Changes to the HLF

In the open source HLF code (commit [46]), the main components responsible for HLF interaction with StateDB have been identified. These are depicted in FIGURE 6. In order to add new StateDBs, it was decided to create an analogue of the marked components for each database. So the main code upgrade was to add three code files for each DB: stateNameOfDB.go, NameOfDB_provider.go, NameOfDB_helper.go. Functionality of goleveldb has been fully preserved. To select StateDB you need to specify in core.yaml file the name of selected DB before running the container.

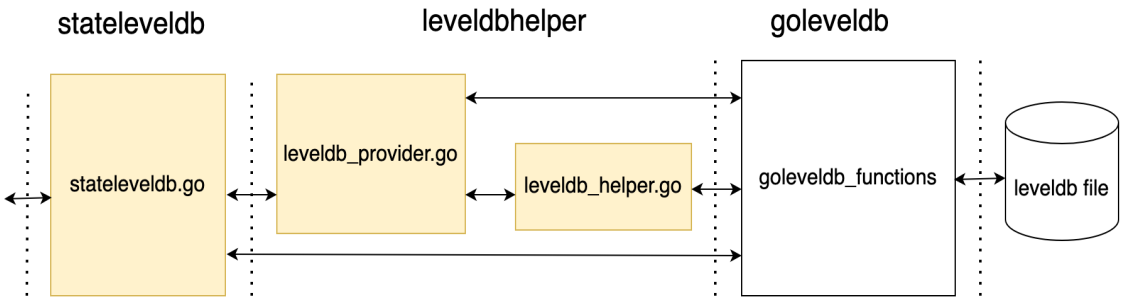

Figure 6: HLF components responsible for interacting with StateDB (highlighted in colour).

Let's walk through the functions and modules that have been added:

5.1.1 stateNameOfDB.go file

- `VersionedDBProvider struct` – satisfies the top-level interface `VersionedDBProvider` and is responsible for top-level DB opening and closing, snapshot import, and logically divides one database into multiple `versionedDB` instances to independently access HLF channels, since channel data does not overlap. It also allows the entire data of a channel to be deleted by calling the `Drop` method. Basically, structure methods work by calling methods from NameOfDB_provider.go file. Although some of the connected databases have built-in logical partition functions, it was decided to keep this functionality at the current level to reduce the embedding time for all StateDBs.
- `versionedDB struct` – satisfies the top-level interface `VersionedDB` and provides many methods that allow access to StateDB. Among them, methods such as: reading values `GetState`, reading versions of values `GetVersion`, opening iterator for range of values `GetStateRange` `ScanIteratorWithPagination` (through initialization object `kvScanner`), writing batch to the database `ApplyUpdates` and others.
- `kvScanner struct` – structure, which is a layer to work with the iterator, provides primarily methods `Next` and `Close`.

5.1.2 NameOfDB_provider.go file

- `Provider struct` – is another layer between HLF and StateDB and is responsible for opening and closing the database, provides separation into multiple logical database partitions `DBHandle`. Using this structure, `versionedDB` instances are created from stateNameOfDB.go file.
- `DBHandle struct` – is the underlying layer through which `versionedDB struct` operates, is an abstraction of the database logical partition and serves methods for reading a value from the database `Get`, writing a value to the database `Put`, writing a batch to the database `WriteBatch` (notable, that `Put` is not used when calling `ApplyUpdates`, only `WriteBatch` is used), delete value `Delete`, delete all data belonging to a certain channel `Drop` and function that opens iterator `GetIterator`.
- `UpdateBatch struct` – temporary storage in memory for the values to be written when the batch is written to the database `WriteBatch`. `UpdateBatch` provides methods to add items to the batch `Put` and remove `Delete`. Because of the different batch handling features in different databases, this structure has been implemented differently for each plugin StateDB.
- `Iterator struct` – provides a top-level interface to work with the database iterator. All embedded databases have different logic of iterator's work – different in `Seek` function, processing of its bounds, errors, etc. Therefore the `Iterator` was upgraded depending on the specifics of each database. This allowed for compatibility with the default StateDB goleveldb.

5.1.3 NameOfDB_helper.go file

- `DB struct` – a structure that holds directly an instance of the opened DB, providing methods for opening the DB `Open`, closing `Close`, reading a value from the DB `Get`, writing a

value to the DB `Put`, removing `Delete`, getting an iterator `GetIterator` and writing a batch `WriteBatch`. As you can see, almost all methods repeat the functionality of the methods of the NameOfDB_provider.go file. This suggests the need to optimize this HLF module, which in order to save time it was decided to include in the future work.

- `FileLock struct` – is responsible for providing a lock to the opened goleveldb database by some process. In general, all embedded databases have such a locking mechanism internally. However it was decided to keep this code for all connected databases to save connection time.

In addition to the modules described, test files covering the added code have also been created and some minor changes have been implemented in some other HLF modules.

5.2 Embedding RocksDB into HLF

To connect RocksDB as StateDB you need to use cgo as the RocksDB library is written in C++. The selected grocksdb API provided the tools to call the C++ functions of the library from the HLF.

In the process, the RocksDB dynamic library was successfully connected.

During the process of RocksDB embedding the unstable work of the iterator in the grocksdb API was found, so it was initiated a patch [47]. Eventually RocksDB v.6.27.3 was embedded to HLF with grocksdb 1.6.45 with the iterator working stably [48].

Additionally, the final docker images of peer and tools, used to run HLF, turned out to be significantly larger than the originals: peer image was larger by 3.4 GB (+6309%), tools image was also larger by 3.4 GB (+847%). This is due to storing auxiliary dynamic libraries in the images. Solving this problem may be an area for further research.

5.3 Embedding bboltdb and BadgerDB into HLF

There are no particular difficulties in embedding bbolt. It was decided to use transactions instead of batches.

When embedding BadgerDB, it was necessary to implement additional methods to get similar logic to the golevedb methods. So, BadgerDB lacks built-in ability to iterate within a given range, ability to iterate back a step, etc. It also required additional runtime checks related to byte-by-byte comparisons of slices, which could affect the performance of iterator operations.

Link to StateDB bbolt repository: [49]; StateDB BadgerDB repository link: [50].

6. BENCHMARKS FOR EMBEDDED STATEDB IN HLF

Performance evaluations of HLF with the new embedded StateDBs were carried out using the Hyperledger Caliper tool[51], which offers universal benchmarks for several blockchain platforms, including Hyperledger Fabric.

This utility supports the capture of basic blockchain performance [52] metrics. Due to the large number of metrics to compare, in order to simplify for better visualization in our work we will consider:

- Write/read transactions throughput,
- Latency (delay in receiving a response),
- CPU usage,
- RAM usage,
- Disk read/write speeds.

The performance benchmarks are based on code samples from [53].

As well as in 2.6, the term ”transaction” is used here in a general sense and refers to any interaction with a smart contract, regardless of the complexity of the subsequent interaction(s) with the HLF platform. The different information content of transactions affects the performance of the HLF network. Therefore, in addition to separating transaction types, which are specified in the headers of the database comparison graphs, transactions have also been divided into subtypes signed on the OX axis. Measuring transaction throughput demonstrated the potential transaction rates and in addition allowed us to track the performance differences for GoLevelDB API calls versus other DB APIs. The data shown in the graphs were measured in the controlled environment. Results obtained in other environments may vary.

The preliminary configuration of the HLF benchmark network was as follows: one channel in which one organization with one peer was deployed and there was another peer of the ordering service. Network endorsement policy: 1-of-any. This simplified configuration was chosen in order to minimize latency from other HLF components and to get as accurate information as possible about components associated only with StateDB.

The results show the average performance of each StateDB.

Since StateDB only runs on an endorsement peer and a validator peer, it was decided to move the peer machine apart from the rest of the HLF network. The configuration of the peer machine on which the performance benchmarks were run is presented in the Table 2. FixedAsset chaincode methods [54] were developed in Golang to evaluate the performance of various interactions with StateDB.

Although there is evidence that write operations need performance improvements the most, we must make sure that read operations performance are not adversely affected. Thus, two kinds of performance benchmarks for read and write operations are solicited for consideration.

Table 2: The peer machine configuration which was used to execute performance benchmarks in Hyperledger Caliper for embedded StateDBs.

OS:	Ubuntu 20.04.4 LTS
CPU(s):	4
RAM:	16
Total SSD memory:	60 GB
Max. bandwidth (read | write):	30 MB/s | 30 MB/s
Max. IOPS (read | write):	2000 | 2000
CPU family:	6
Model:	106
Model name:	Intel Xeon Processor (Icelake)
Thread(s) per core:	2
Core(s) per socket:	2
Socket(s):	1

6.1 Get-Asset Performance Benchmark

The scenario under study is aimed at reading from StateDB. The read performance benchmark consists of Evaluate calls to the GetAsset method of the FixedAsset chaincode. This chaincode was deployed in independent HLF networks per each considered StateDB. Each network has its own database as StateDB. One of them has the standard goleveldb and others have one of the proposed alternatives: RocksDB, bboltdb or BadgerDB.

Every network uses a ”1-of-any” endorsment policy to minimize latencies (there are one peer and one orderer deployed in the network. The peer is acting as an endorser and committer) The chaincode method runs on a single Hyperledger Fabric peer and does not result in any interaction with the orderer. Resulted transaction pathway of this performance benchmark is depicted in FIGURE 7.

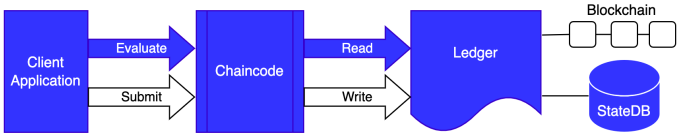

Figure 7: Diagram of HLF components interaction during Get-Asset-Evaluate benchmark. Active components are highlighted in blue.

10 clients interact with the network. Before starting the performance benchmark, StateDB is initialized - each client sends a command to create (write) 1000 values of each type: 100 bytes / 1000 bytes / 2000 bytes / 4000 bytes / 8000 bytes / 16000 bytes / 32000 bytes / 64000 bytes. The key name corresponding to value contains a prefix relating to each type of value and the number of the sending client (for example, for client 5 and value type 4000 bytes, the prefix will be: ”client5_4000_”). The prefix is followed by a key id in the range from 1 to 1000.

Then a separate performance sub-benchmark is run for each type of value, lasting 5 minutes. 10 clients send transactions concurrently to retrieve one type of asset with a random ID from StateDB. Achievable throughput is investigated through maintaining a constant transaction backlog of 50 transactions for each of the 10 test clients.

FIGURE 8 shows a comparison of transaction throughput for embedded StateDBs. It demonstrates that the alternative StateDBs didn't see any appreciable performance reduction. In every instance, BadgerDB has an even somewhat higher TPS than the original goleveldb. At the same time, all StateDBs' measured Latency rates displayed about the same numbers (we will not give here). What about resource consumption, though?

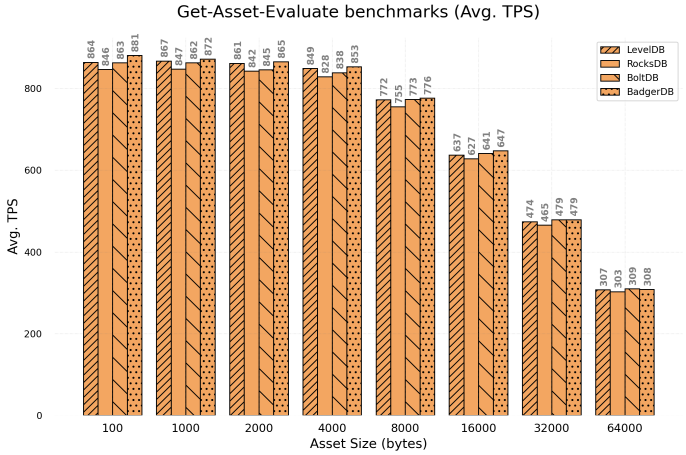

Figure 8: Get-Asset performance benchmark results (transactions per second rate).

Likewise, all StateDBs' CPU utilization rates were found to range between 23% and 32%. (we will not illustrate the CPU graph here). But with RAM consumption, things are more interesting - see FIGURE 9. It becomes apparent that, when compared to goleveldb and bbolt, RocksDB and BadgerDB require excessively more RAM. Let's leave the facts as they are for now and draw conclusions afterwards.

6.2 Create-Asset performance benchmark

The scenario under study is aimed at writing to StateDB. The write performance benchmark consists of Submit calls to the CreateAsset method of the FixedAsset chaincode. This chaincode was deployed in independent HLF networks per each considered StateDB. Each network has its own database as StateDB. One of them has the standard goleveldb and others have one of the proposed alternatives: RocksDB, bboltdb or BadgerDB.

Every network uses a "1-of-any" endorsment policy to minimize latencies (The network has one peer and one orderer deployed. The peer is acting as an endorser and committer) The chaincode method runs on the endorser and its result is then placed on the ledger by passing through the orderer and the committer. Resulted transaction pathway of this performance benchmark is depicted in FIGURE 10.

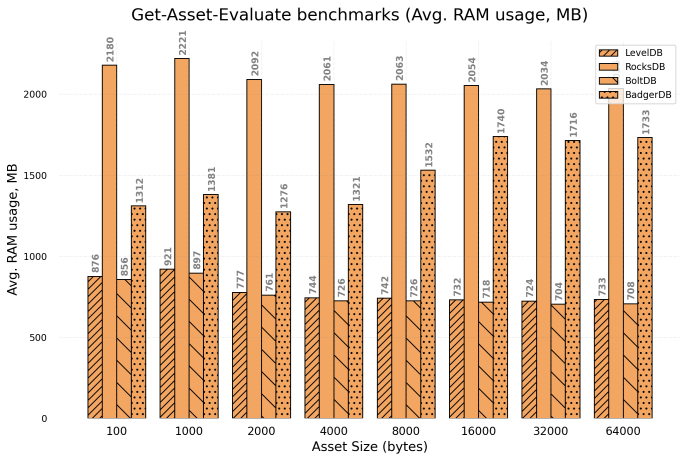

Figure 9: Get-Asset performance benchmark results (average RAM consumption score).

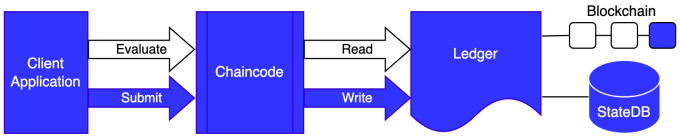

Figure 10: Diagram of HLF components interaction during Create-Asset-Submit benchmark. Active components are highlighted in blue.

There are 10 clients interacting with the network. At the beginning of the performance benchmark, no initialization occurs. As soon as HLF starts, performance readings begin to be collected for StateDB write transactions. Concurrently with the other clients, every client submits transactions to the endorser. Each transaction writes one key-value pair to the StateDB. It lasts 5 minutes for each client for each type of transaction: 100 bytes / 1000 bytes / 4000 bytes / 8000 bytes / 16000 bytes / 24000 bytes / 32000 bytes / 64000 bytes. Each key of the written value consists of: client number, value size and transaction sequence number (for example, for client 3 and value type 24000 bytes, the key for the seventh transaction sent by the client would be: ”client3_24000_7”). Achievable throughput is investigated through maintaining a constant transaction backlog of 10 transactions for each of the 10 test clients.

FIGURE 11 compares the throughput of HLF transactions for embedded StateDBs. It shows that all StateDB compete with each other almost on an equal footing. However, StateDB BadgerDB has a slight TPS edge for all types of transactions. Additionally, for a transaction size of 64KB, BadgerDB’s TPS is much greater than goleveldb’s. In turn, RocksDB showed equal performance with goleveldb and achieved a clear advantage for a transaction size of 64KB. Similar to this, bbolt only demonstrated a marginal benefit over goleveldb for transactions of 32KB and 64KB in size.

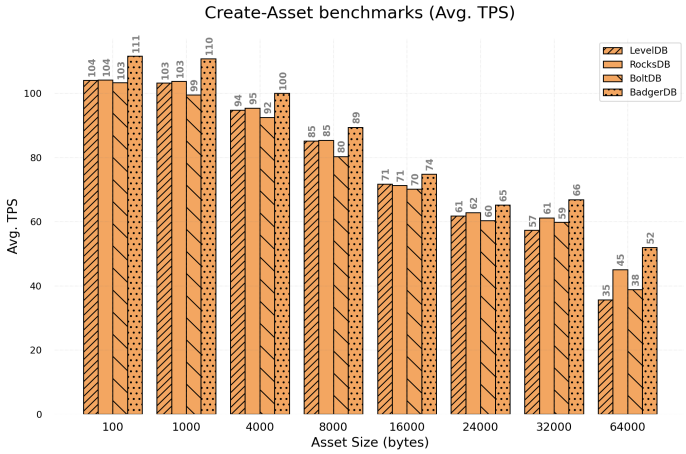

Figure 11: Create-Asset performance benchmark results (transactions per second rate).

The Latency comparison graph in FIGURE 12, confirms that BadgerDB is more efficient than other StateDBs.

The CPU use comparison in FIGURE 13, further demonstrates that alternative StateDBs are in a competitive position with goleveldb. But the RAM utilization graph (FIGURE 14) once more demonstrates the increasing memory usage of RocksDB and BadgerDB. The speed of writing to the disk of the peer was higher for goleveldb and bbolt (FIGURE 15).

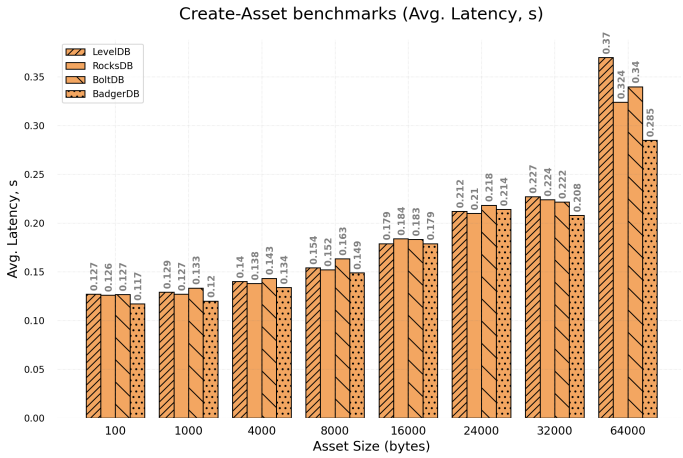

Figure 12: Create-Asset performance benchmark results (average Latency rate).

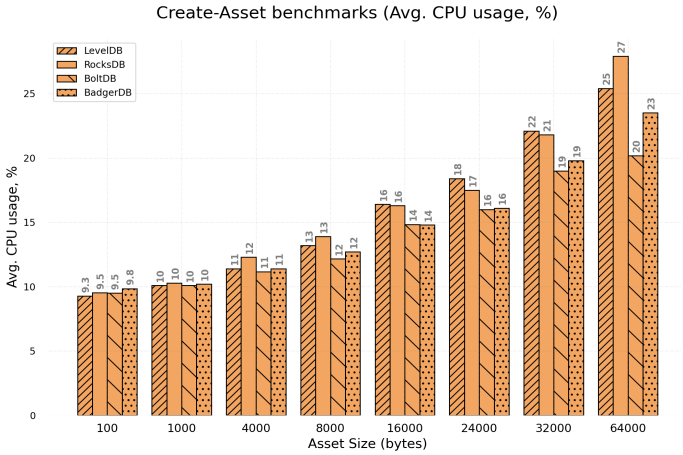

Figure 13: Create-Asset performance benchmark results (average CPU consumption rate).

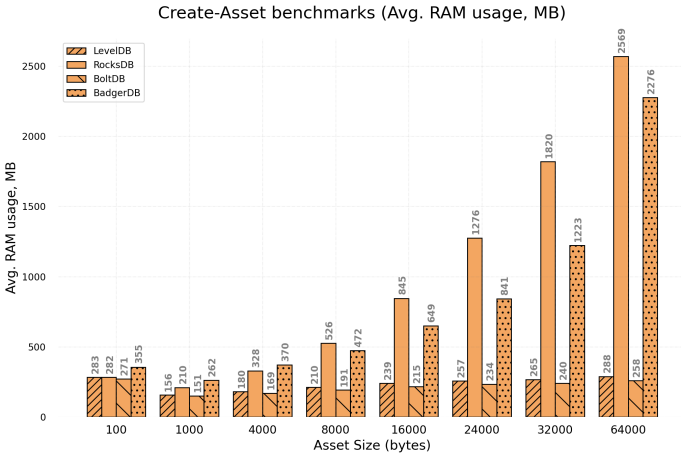

Figure 14: Create-Asset performance benchmark results (average RAM usage rate).

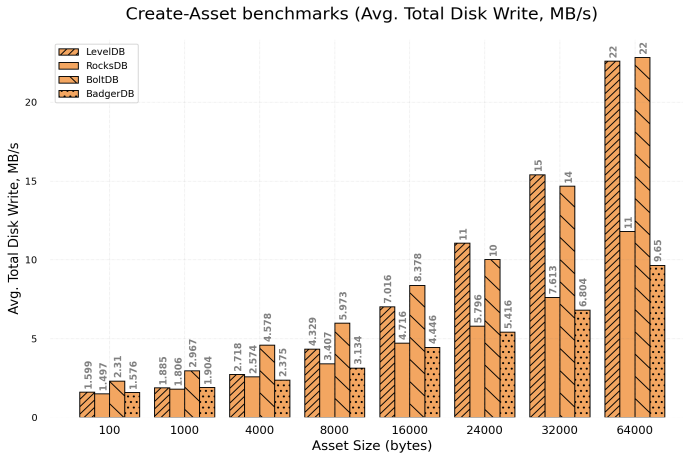

Figure 15: Create-Asset performance benchmark results (average Disk Writes per second rate).

7. ANALYSIS OF COLLECTED PERFORMANCE DATA

7.1 RAM

Get-Asset generally showed higher RAM consumption than Create-Asset for all StateDBs (FIGURE 9 and FIGURE 14). It is clear that RocksDB and BadgerDB require more RAM at the current settings of both StateDBs than their competitors – goleveldb and bbolt.

Let's try to understand the reason for BadgerDB's increased RAM usage. For the same total (alloc_space) bytes in the HLF with goleveldb and in the HLF with BadgerDB, we compared directly the allocated volume of each StateDB. As expected for BadgerDB, this volume was significantly larger: with the total same allocated bytes of the two HLF-StateDB networks, BadgerDB showed a total allocated volume of 5GB more than goleveldb. This indicates that BadgerDB requires more RAM to run.

BadgerDB functions showed increased allocations:

- badger.(*DB).flushMemtable,
- table.(*Builder).handleBlock,
- badger.(*levelsController).subcompact,
- badger.(*DB).writeRequests.

The amount of memory used in the moment (inuse_space) for BadgerDB was also higher by 300MB. BadgerDB functions that used memory:

- badger.(*DB).flushMemtable,
- badger.(*DB).doWrites,

- table.(*Builder).handleBlock.

Overall, there is no huge difference between total inuse_space, which suggests also some hidden use of RAM in BadgerDB. An analysis of online discussions on this topic reveals potentially several reasons:

- Using BadgerDB's default options, which can cause an increased cache size, for example. A list of options which affect RAM is given in [55], which can also affect the amount of total bytes allocated;
- The lack of the automatic garbage collector in Golang in BadgerDB [55].

7.2 Disk Writes for Create-Asset benchmark

The average write speeds shown in FIGURE 15 are well below the maximum possible write speeds for this machine configuration (see Table 2). Analysing the metrics in grafana [56], it was found that write speeds for all databases periodically reach their maximum value, but only for transaction sizes of 24KB and above. Their average write speed, however, is still lower than the maximum possible speed due to constant speed bumps – see FIGURE 16.

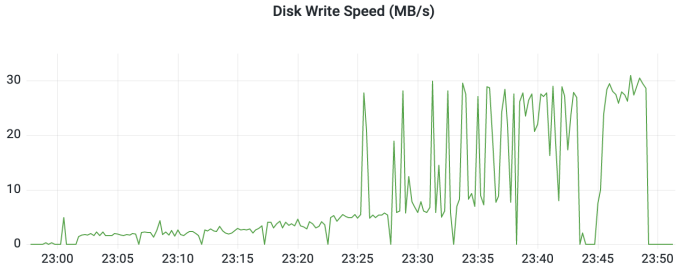

Figure 16: The sequential change in disk write speed over time for each type of value in the Create-Asset performance benchmark for one of the loaded StateDBs. From 23:01 and onwards in 5 min intervals the values 100, 1000, 4000, 8000, 16000, 24000, 32000, 64000 bytes were written to the StateDB in turn.

As already mentioned in 3, the efficient use of HLF resources leaves a lot to be desired, so there is not a 100% load on disk and CPU consumption for any StateDB. On the other hand, the presence of an abundance of locks is also the reason for not 100% CPU utilization.

8. RELATED WORK

Several studies have investigated the bottlenecks in Hyperledger Fabric with a specific focus on enhancing scalability and optimizing the performance of the state database [57]. Thakkar et al. [16] conducted an extensive analysis of configurable parameters and proposed optimizations to improve throughput and transaction latency [58, 59]. They identified significant bottlenecks such

as crypto operations, transaction validation, and multiple API calls to CouchDB. Nakaike et al. [60] focused on the performance analysis of database systems used in Hyperledger Fabric, particularly GoLevelDB. Their investigation revealed data compression and database size as major performance bottlenecks. Additionally, accessing multiple values split from a single large value was observed to be slower than accessing the single large value.

In a comprehensive analysis of the execute-order-validate transaction model in Hyperledger Fabric v1.1, the authors in [61] examined the system's scalability and performance. They highlighted the communication overhead between the execution and ordering phase as a critical bottleneck and demonstrated the impact of scaling the Kafka cluster used for ordering. Scalability analysis of endorsing peers, ordering service nodes, and the performance characteristics of each phase of the transaction life cycle were investigated in [62, 63]. The study [62] compared the performance of three ordering services (Kafka, Raft, and Solo) and identified the validation phase as the system bottleneck due to slow validation speeds.

To evaluate the performance impact of migrating classic workload-oriented applications to blockchain-based distributed ledger technologies, a study by the authors in [64] introduced a full port of the TPC-C benchmark to Hyperledger Fabric. The research emphasized the need for application-level performance benchmarks and presented an open implementation of the TPC-C OLTP benchmark, addressing the lack of comprehensive macrobenchmarks and workload composition considerations.

Collectively, these studies contribute to the advancement of Hyperledger Fabric's performance optimization. They offer valuable insights into parameter configuration, database performance, transaction model analysis, scalability evaluation, and benchmarking approaches, facilitating the development of efficient and scalable Hyperledger Fabric solutions for enterprise applications. However, none of the studies attempted to rethink and replace the state database with a different database engine to examine its impact on the overall performance of Hyperledger Fabric.

9. CONCLUSIONS AND FURTHER DIRECTION OF WORK

As a result of the research, several databases competing with goleveldb in terms of performance and functionality were selected. The selected databases were embedded in HLF as StateDB. While investigating HLF resource consumption, one of the reasons slowing down HLF with the default of StateDB was discovered and eliminated. Furthermore, performance benchmarks were created and conducted for each of the possible alternative StateDBs. In the end, a favorite of the potential StateDBs was identified - BadgerDB.

BadgerDB demonstrated high performance results in the reviewed published papers, had a slight advantage in the StateDB read performance benchmarks, and showed a decent advantage in the StateDB write performance benchmarks. BadgerDB had the strongest advantage over goleveldb for write values of 64KB (TPS is 1.5 times higher). In addition, BadgerDB provides StateDB with features that were not implemented with goleveldb: it guarantees ACID properties and makes it possible to use custom queries in HLF by implementing an additional tool [30].

The HLF source code with linked databases is available in the repositories: [48] (RocksDB), [49] (bbolt), [50] (BadgerDB).

To run the performance benchmarks, the HLF fabric-samples/test-network was modified to provide the minimum required set of network components in order to test the performance of StateDB peer with the new databases `https://github.com/fubss/fabric-samples/tree/dbs_selector`.

Based on the Hyperledger Caliper [53] demo, a custom repository was created for the HLF load, adapted for OS Linux and OS X (in different branches) https://github.com/fubss/caliper-workspace-3.

In further research, it is suggested that:

- Identify and eliminate other HLF components that slow it down and eliminate repetitive code fragments;
- Eliminate the increased RAM usage of StateDB BadgerDB by finding the right configuration with code changes;
- Implement an additional tool for StateDB BadgerDB allowing custom queries to the StateDB.

10. ACKNOWLEDGMENT

We thank Vladimir Chechetkin for embedding BadgerDB and other contributions to the work on this paper.

References

[1] Manevich Y, Barger A, Assa G. Redacting Transactions From Execute-Order-Validate Blockchains. In: IEEE International Conference on Blockchain and Cryptocurrency (ICBC). 2021:1-9.

[2] Abou Jaoude J, Saade RG. Blockchain Applications – Usage in Different Domains. IEEE Access. 2019;7:45360-45381.

[3] Nor SM, Abdul-Majid M, Esrati SN. The Role of Blockchain Technology in Enhancing Islamic Social Finance: The Case of Zakah Management in Malaysia. foresight. 2021;23:509-527.

[4] Fedorov IR, Pimenov AV, Panin GA, Bezzateev SV. Blockchain in 5-G Networks: Perfomance Evaluation of Private Blockchain. In: Wave Electronics and Its Application in Information and Telecommunication Systems (Weconf). IEEE Publications. 2021:1-4.

[5] https://hyperledger-fabric.readthedocs.io/en/latest/

[6] https://www.hyperledger.org/learn/case-studies

[7] Barger A, Ilina O, Zemtsov A, Tagirova K. Trustful Charity Foundation Platform Based on Hyperledger Fabric. In: IEEE International conference on omni-layer intelligent systems (COINS). IEEE Publications. 2022:1-6.

[8] https://www.hyperledger.org/use/fabric

[9] Androulaki E, Barger A, Bortnikov V, Cachin C, Christidis K, et al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In: Proceedings of the thirteenth EuroSys conference. 2018:1-15.

[10] Barger A, Manevich Y, Meir H, Tock Y. A Byzantine Faulttolerant Consensus Library for Hyperledger Fabric. In: IEEE International conference on blockchain and cryptocurrency (ICBC).2021:1-9.

[11] Papadimitriou CH, Kanellakis PC. On Concurrency Control by Multiple Versions. ACM Trans Database Syst. 1984;9:89-99.

[12] Nakaike T, Zhang Q, Ueda Y, Inagaki T, Ohara M, et al. Hyperledger Fabric Performance Characterization and Optimization Using Goleveldb Benchmark. In: IEEE International Conference on Blockchain and Cryptocurrency (ICBC). 2020:1-9.

[13] https://github.com/syndtr/goleveldb

[14] https://lists.hyperledger.org/g/fabric/message/10357

[15] https://wiki.hyperledger.org/display/fabric/Fabric+Strategic+Priorities+-+2021+discussion

[16] Thakkar P, Nathan S, Viswanathan B. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. In: 26th international symposium on modeling, analysis, and simulation of computer and telecommunication systems (MASCOTS). IEEE Publications. 2018:264-276.

[17] https://hyperledger.github.io/caliper-benchmarks/fabric/performance/2.1.0/nodeContract/nodeSDK/configuration/

[18] https://pkg.go.dev/net/http/pprof

[19] https://github.com/felixge/fgprof

[20] https://github.com/felixge/fgtrace

[21] https://github.com/hyperledger/fabric/commit/f17d1d934ca2f65740561f06277b90c7eee5fa89

[22] https://github.com/hyperledger/fabric/compare/main...fubss:fabric:leveldb-tune-4

[23] https://couchdb.apache.org/

[24] Podgorelec B, Turkanovic M, Sestak M. A brief review of database solutions used within blockchain platforms. In International Congress on Blockchain and Applications. Springer; 2020:121-130.

[25] https://github.com/facebook/rocksdb

[26] https://github.com/linxGnu/grocksdb

[27] https://github.com/smallnest/kvbench

[28] https://github.com/facebook/rocksdb/wiki/Features-Not-in-LevelDB

[29] https://github.com/asdine/storm

[30] https://github.com/timshannon/badgerhold

[31] Henry G. Howard Chu on Lightning Memory-Mapped Database. IEEE Softw. 2019;36:83-87.

[32] https://github.com/LMDB/lmdb

[33] https://github.com/bmatsuo/lmdb-go

[34] https://github.com/boltdb/bolt

[35] https://github.com/etcd-io/bbolt

[36] https://medium.com/@smallnest/go-k-v-databases-benchmark-cd051279ef22

[37] https://github.com/akrylysov/pogreb

[38] https://github.com/tidwall/buntdb

[39] https://github.com/nutsdb/nutsdb

[40] https://github.com/dgraph-io/badger

[41] https://github.com/cockroachdb/pebble

[42] https://dgraph.io/blog/post/badger-lmdb-boltdb/

[43] https://dgraph.io/blog/post/badger/

[44] https://golangexample.com/rosedb-an-embedded-and-fast-k-v-database-based-on-lsm-wal/

[45] https://github.com/flower-corp/rosedb

[46] https://github.com/hyperledger/fabric/commit/6656f72563c73a806dee7068dd91b3acf2a286aa

[47] https://github.com/linxGnu/grocksdb/issues/62

[48] https://github.com/fubss/fabric/tree/grocksdb-30

[49] https://github.com/fubss/fabric/tree/bbolt-30

[50] https://github.com/fubss/fabric/tree/badger-30

[51] https://github.com/hyperledger/caliper

[52] https://www.hyperledger.org/learn/publications/blockchain-performance-metrics#definitions

[53] https://github.com/hyperledger/caliper-benchmarks

[54] https://github.com/fubss/caliper-workspace-3/blob/main/smart-contract/go/FixedAssetContract.go

[55] https://dgraph.io/docs/badger/get-started

[56] https://grafana.com/

[57] Nasir Q, Qasse IA, Abu Talib M, Nassif AB. Performance Analysis of Hyperledger Fabric Platforms. Secur Commun Netw. 2018:2018:1-14.

[58] Sukhwani H, Wang N, Trivedi KS, Rindos A. Performance Modeling of Hyperledger Fabric (Permissioned Blockchain Network). IEEE 17th International Symposium on Network Computing and Applications (NCA). IEEE Publications. 2018:1-8.

[59] Swathi P, Venkatesan M. Scalability Improvement and Analysis of Permissioned-Blockchain. ICT Express. 2021;7:283-289.

[60] Nakaike T, Zhang Q, Ueda Y, Inagaki T, Ohara M, et al. Hyperledger Fabric Performance Characterization and Optimization Using Goleveldb Benchmark. In: IEEE International Conference on Blockchain and Cryptocurrency (ICBC). 2020:1-9.

[61] Nguyen MQ, Loghin D, Dinh TTA. 'Understanding the Scalability of Hyperledger Fabric. 2021. Arxiv preprint:https://arxiv.org/pdf/2107.09886.pdf

[62] Wang C, Chu X. Performance Characterization and Bottleneck Analysis of Hyperledger Fabric. In 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS) IEEE Publications. 2020:1281-1286.

[63] Foschini L, Gavagna A, Martuscelli G, Montanari R. Hyperledger Fabric Blockchain: Chaincode Performance Analysis. ICC 2020-2020 IEEE International Conference on Communications (ICC). 2020:1-6.

[64] Klenik A, Kocsis I. Porting a Benchmark With a Classic Workload to Blockchain: Tpc-C on Hyperledger Fabric. In: Proceedings of the 37^{th} ACM/SIGAPP symposium on applied computing. 2022:290-298.